Tracing the Way of Data in a TCP Connection through the Linux Kernel

Seminar Organic Computing

Richard Sailer
Matrikelnummer: 1192352

Universität Augsburg
Lehrstuhl für Organic Computing
richard.willi.sailer@student.uni-augsburg.de

Abstract

Books on Linux kernel programming grow old quite fast since the Linux Kernel is a very active project. Additionally most of them are quite expensive or difficult to obtain, and have either a quite broad or very focused character, making them less valuable for a aspiring and learning kernel developer.

This seminar paper outlines the usage of ftrace a tracing framework to analyse a running Linux system. Having obtained a trace-log a kernel hacker can read and understand source code more determined and with context.

In a detailed example this approach is demonstrated in tracing and the way of data in a TCP Connection through the kernel. Finally this trace-log is used as base for more a exact conceptual exploration and description of the Linux TCP/IP implementation.

1 Motivation and Introduction

Linux kernel programming is complex and difficult for most students or people with experience in application programming. The C programming language lacks many of the high level features these people are used to and even common C libraries are not accessible in kernel code, actually no C libraries at all is available in kernel space[4].

Additionally the kernel is a complex piece of software consisting of many modules working together in non-trivial ways. To change or extend a part of the kernel semantic knowledge on how and why this parts work together is necessary.

While the first two obstacles, the often unfamiliar C programming language and the lack of libraries, can be overcome with some experience and get smaller and smaller after some time, the possibility of understanding the in-kernel mechanics is an question of good documentation.

While there exist several good books on this topic, all of them have one of the following three problems:

1. They grow old very fast

   Or put another way: The Linux kernel evolves too fast. For example UNDERSTANDING LINUX NETWORK INTERNALS by Christian Benvenuti[1], a really comprehensive books, was written in 2006 covering Linux kernel 2.6. Since it took some time writing some examples and parts cover even older parts, like the bottom halve interrupt handling. Many details in the TCP stack or the interrupt handling have changed since. At the time of writing the current version of the Linux kernel is 4.3

2. They have a certain focus and do preselection of content

   Since the Linux kernel is a very large project consisting of over 19 Million lines of code[1] obviously a preselection is necessary even if the author is focusing on a subsystem. For example in "Linux Kernel Networking" by Rami Rosen[6] TCP is covered quite brief since the lower networking layers get a much deeper coverage.

3. They are not freely available

   Most of these books have to be purchased and ordered. Some of them are quite expensive and ordering them can take several days. These two facts make it more difficult for interested programmers to start Linux kernel programming.

This seminar paper tries to solve two of these three issues. In this paper we will try to provide an free (licensed under the GNU Free Documentation License) and up to date overview of one very specific topic: The Way of Data in a TCP Connection through the Linux kernel. Additionally we will introduce and explain the method used to examine and understand these kernel mechanics in order to give every reader a tool helpful in understanding other parts of the kernel his or herself.

1. Version 4.1, see http://www.phoronix.com/scan.php?page=news_item&px=Linux-19.5M-Stats for details.

2 State of Research and Related Work

The following two papers and Linux kernel documentation documents cover related topics.

2.1 The performance analysis of Linux networking--packet receiving[9]

This work done in 2006 by Wenji Wu and Matt Crawford from the fermilab in Illinois, USA focuses entirely on the packet receive process and performance issues. While providing many valuable insights for performance engineering and answering some on the "tracing the way of data" questions on this work it barely covers the "which function are involved and what are they doing"-question, which is part of the focus in this work, since their primary target was performance optimisation. Also it contains the following diagram of the buffer structure, which is quite helpful for getting an overview and context for the results of the next chapters:

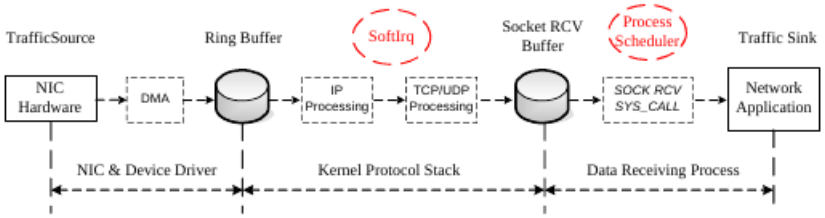

Figure 1. Buffers and Copying in the Linux Kernel

The 2 Buffers shown in Figure 1 will appear again in the results section.

2.2 The "kernel_flow" article in the official Linux Foundation Documentation[4]

In 2009 the Linux foundation released this documentation for the Linux kernel networking stack together with a quite tall and comprehensive diagram (which for layouting and scope reasons is not included in this paper). With a size of 3489x1952 pixels even on a full HD monitor it's not possible to look at the full diagram. Besides it suffers from "putting absolutely everything in one picture" which makes it difficult to get an overview. While being an primary and valuable source for this work it was a goal of this paper to produce an up-to-date and simplified version of this diagram as poster. Simplified in this case means, split into two distinct diagrams one for the receive path and one for the send path.

4. **See: http://www.linuxfoundation.org/collaborate/workgroups/networking/kernel_flow or use pdf-href**

3 About the Measuring Method: ftrace

3.1 About Tracing and Ftrace

Ftrace is a kernel-built-in tracer for function calls and events inside the Linux kernel.[5]

Definition. *tracer (software engineering) [3]*
A tracer is a tool for analysing the behaviour of a given software at runtime. It provides an output similar to an log of what happens inside the program. In most cases this output is an sequential structured list of all function calls which happened during execution. But there also exist other kinds of tracing like events tracing or I/O tracing. For VM-languages like Java or C# tracing is an feature of the runtime environment which can be turned on or off at runtime. For compiled languages like C tracing support has to be added via compiler switches or additional modules.

In the case of ftrace the tracing support is part of the software, the Linux kernel. On most architectures ftrace uses hardware support for better performance.[2]

3.2 A Short Overview of ftrace Capabilities and Usage

Ftrace can be used and controlled via the *trace-cmd* program. *trace-cmd* is packaged and available in all big Linux distributions.[5] Since tracing in kernel operation is still a quite major intervention into a running system only the root user is allowed to use trace-cmd, so all the following examples have to be executed as root.

To simply start recording all the function calls happening in the Linux without any filtering use:

```
trace-cmd record -p function_graph
```

Figure 2. Example: Start tracing of all function calls in Linux kernel [7]

This writes all the results into a trace.dat in your working directory. `record` is one of the several sub-commands of trace-cmd, in our examples and later measurements only the `record` and the `report` sub-commands are needed. You shouldn't run this command (in the unfiltered version) too long, since it produces quite big files, about 900 MB after 30 seconds of tracing appeared in all tests using unfiltered tracing.

5. At the time of this writing (03.01.2016) *trace-cmd* is available in Ubuntu (since 12.04), Debian testing and stable and Fedora.

To view the content of the `trace.dat` file in a human readable format use:

```
trace-cmd report > results
```

Figure 3. Converting the Results into an human readable format.

The report sub-commands automatically uses the trace.dat file in the working directory and writes it's content to STDOUT which the redirection operator redirects to the `results` file.

The human readable output contains several columns about: the name of the process on behalf of the in kernel function call happened, the id of the CPU, an absolute time stamp, info if it's a function exit or entry event, the time the function needed (most below one micro-second) and the function name. The function names are graphically indented to display the call hierarchy, so if B is called by A, B is indented relative to A by 2 spaces.

Usually these are much more columns than needed so in the results you will see in these paper, some of these columns were removed for layouting reasons.

3.3 Filtering

For analysing the way of data of a TCP connection we do not need information of all functions called in the overall kernel. We're only interested in tracing of the function calls happening on behalf of one single application. Kernel side filtering after a specific pid is possible using:

```
trace-cmd -p function_graph -P <pid>
```

Figure 4. Tracing all in-kernel function calls happening on behalf of <pid>[7]

This way, the log file sizes are much smaller and more focused than previously. Tracing netcat for some time, while sending and receiving 3 small text messages produced a trace log file of 2,3 MB.

4 Test Setup and Results

4.1 Test Setup

To produce measurable network traffic the *BSD netcat* program has been used. Netcat is a small Unix command line program which opens a TCP (or UDP or Unix Domain) Socket, either as listening socket, or as "client" to connect to another socket. The IP and Port to connect to (or the port to listen on)

are supplied as command line parameters. For example: `nc 17.17.17.17 1055` connects to the IP 17.17.17.17 on port 1055 via an IPv4 TCP connection. Complementary with `nc -l 1055` the process opens an listening socket on the local machine on port 1055, waiting for a IPv4 TCP connection. After establishing a connection netcat sends all data it gets from STDIN through the socket and prints all data it receives through the socket to STDOUT.

For this experiment 2 netcat instances were used, one on the measurement computer another on an remote Linux server. The command used on the measurement computer was `nc -l 1337` and the remote server connected via `nc <ip> 1337`. After the connection was established in another terminal tracing was started using:

```
trace-cmd record -p function_graph -P <nc-pid>
```

Then two short messages (strings of 9 Byte and 167 Byte) were sent from the measurement computer. Following two messages of equal size were sent from the remote server and received by the measurement computer. Finalising the tracing was stopped and the results translated into a human readable file using `trace-cmd report`.

4.2 Test results

Since the full trace of all function calls happening on behalf of netcat contained about 3000 lines (which subtracting all the empty lines drawn for the ASCII art graph and closing brackets are about 2100 function calls), post editing got necessary. Most of the function calls involved scheduling, terminal I/O or kernel internal locking of resources, so the sequences belonging to sending or receiving one packet were located and extracted. This happened by following the `SyS_write()` and `sys_read()` calls, which are the syscalls netcat uses to send and receive packets. This was gathered through tracing all the syscalls netcat does using strace.

The receive sequence consisted of 37 function calls and 56 lines which is small enough to include the complete trace in this document. Contrastingly the send sequence comprised 510 lines, so shortening got necessary. The shortening included removing most of the locking and mutex function calls. Also many cases in which `function()` , did some locking and then called `__function()` for doing the internal work were simplified by only keeping the `function()` call. As a last step, the indentation and superfluous columns of both results were removed, so both traces fit into this document side by side.

The final simplified traces are visible in Figure 5 and Figure 6. The full and unedited trace results are available via [6] and [7].

6. github: https://github.com/richi235/ftrace-seminar-paper

7. archive.org: https://archive.org/details/final_trace_03.01.tar as archive

```
SyS_write() {
__fdget_pos() [...]
vfs_write() {
rw_verify_area() [...]
__vfs_write() {
sock_write_iter() {
sock_sendmsg() {
inet_sendmsg() {
tcp_sendmsg() {
lock_sock_nested() [...]
tcp_send_mss() [...]
sk_stream_alloc_skb() [...]
skb_entail() [...]
skb_put();
tcp_push() {
__tcp_push_pending_frames() {
tcp_write_xmit() {
tcp_init_tso_segs();
tcp_transmit_skb() {
skb_clone() [...]
skb_push();
tcp_v4_send_check() [...]
bictcp_cwnd_event();
ip_queue_xmit() {
skb_push();
ip_local_out_sk() {
__ip_local_out_sk() {
ip_send_check();
nf_hook_slow() [...]
ip_output() {
nf_hook_slow() [...]
ip_finish_output() {
ip_finish_output2() {
skb_push();
dev_queue_xmit_sk() {
__dev_queue_xmit() {
skb_clone() {
kmem_cache_alloc();
__skb_clone() {
__copy_skb_header();
skb_release_all() {
skb_release_head_state();
skb_release_data();
kfree_skbmem()
kmem_cache_free();
e1000_xmit_frame() [...]
} } } } } } }
tcp_event_new_data_sent() {
tcp_rearm_rto() {
tcp_rearm_rto.part.59() {
sk_reset_timer() {
mod_timer() [...]
} } } } } }
release_sock() {
} } } }
fsnotify();
} }
```

Figure 5. Sending a TCP packet, simplified kernel trace result

```
sys_read() {
__fdget_pos() {
__fget_light();
}
vfs_read() {
rw_verify_area() {
security_file_permission() {
__fsnotify_parent();
fsnotify();
}
}
__vfs_read() {
sock_read_iter() {
sock_recvmsg() {
security_socket_recvmsg();
inet_recvmsg() {
tcp_recvmsg() {
lock_sock_nested() {
_cond_resched();
_raw_spin_lock_bh();
__local_bh_enable_ip();
}
skb_copy_datagram_iter();
tcp_rcv_space_adjust();
__kfree_skb() {
skb_release_all() {
skb_release_head_state() {
sock_rfree();
}
skb_release_data() {
kfree();
}
}
kfree_skbmem() {
kmem_cache_free();
}
}
tcp_cleanup_rbuf() {
__tcp_select_window();
}
release_sock() {
_raw_spin_lock_bh();
tcp_release_cb();
_raw_spin_unlock_bh() {
__local_bh_enable_ip();
}
}
}
}
}
}
}
__fsnotify_parent();
fsnotify();
}
}
```

Figure 6. Receiving a TCP packet complete kernel trace results

5 Evaluation and Discussion of the Trace Results

Since this trace log alone is only semi-illuminating, in the following section we will explain what happens during these function calls, to give some overview. If not noted otherwise the source for these descriptions are the function definitions in the Linux source code in the stable version 4.3.1.

5.1 Send Flow

For a better understanding the whole TCP send sequence can be split up into 5 sub-sequences which we will cover and explain separately. The 5 sequences are: (1) *copy-to-kernelspace*, (2) *tcp-processing*, (3) *ip-and-netfilter-processing*, (4) *ethernet-and-driver-processing*, (5) *finishing-works*.

copy-to-kernelspace: This happens between the invocation of the `write()`-syscall in the user space and the in kernel function `sock_sendmsg()`. `SyS_write()` handles the write() system-call from user space and initiates the copying of the data in the user space buffer to the kernelspace. This copying is a expensive operation[6].

tcp-processing: This happens between `tcp_sendmsg()` and `ip_queue_xmit()`. Here the check-sums are calculated and the TCP Header is built, this all happens using the *socket buffer skb* the central data structure of Linux networking. For the *skb* there exist several utility functions like `skb_clone()` or `skb_push()` it's important to understand, that `skb_clone()` does not copy the entire data structure but only header and management info, since the data is referenced thorough a pointer, so only this pointer gets copied.

ip-and-netfilter-processing: This happens between `ip_queue_xmit()` and `ip_finish_output2()`. Here several things happen: (1) The IP packet and header is built, (2) The routing decision is made, (3) Firewall and forwarding rules of netfilter are applied (This happens in `nf_hook_slow()` and involves many function calls, most of them where cut out because netfilter is beyond the scope of this work).

ethernet-and-driver-processing: This happens between `ip_finish_output2()` and `e1000_xmit_frame()`. At first the whole socket buffer is copied to the network interface (see [9] for a detailed description of this process). It depends on the network device and the driver if the device has a own buffer or uses the main memory via DMA. [9] [6] Then the *skb* used by the IP sub-sequence gets freed using `skb_release_all()` and finally `kfree_skbmem()`. Finally the packet is transmitted using `e1000_xmit_frame()`, which is a device-driver specific function of the Intel e1000 network card of the test computer.

finishing-works: This happens between `tcp_event_new_data_sent()` and `fsnotify()`. Remember that this is the path of a successful transmission, the `tcp_rearm_rto()` function is only called after an ACK packet is received. Here the *Retransmission Timeout* (*RTO*) and the *Congestion Window* (cwnd) get adjusted according to the successful transmission, for details of the TCP logic see the TCP section of [8]. At last the filesystem is notified since on Linux there exists an inode for every tcp-socket (in a own namespace tough).

5.2 Receive Flow

The receive sequence is much more overseeable, here the fragmentation in two sub-sequences suffices for understanding. They are the (1) *copy-to-userspace* and the (2) *tcp-processing* sequences.

copy-to-userspace: This happens before and after the *tcp-processing* sequence, it can be said the *copy-to-userspace* section is split up, with the *tcp-processing* sequence between. The first slice starts from the systemcall in userspace end ends at `sock_read_iter()`. The second slice starts at `tcp_release_sock()` and ends with `fsnotify()`. This sequence first allocates data structures for the data and manages locking, then fetches the data form TCP and finally copies the data to userspace, de-allocates the data structures used and notices the filesystem, comparable to the send flow.

tcp-processing: This happens between `sock_recvmsg()` and `tcp_cleanup_rbuf()`. Here pointers to the data are copied from the *socket buffer* handled by the tcp stack, and finally the *skb* of the tcp internals gets freed. Additionally the `tcp_rcv_space_adjust()` call updates the `rmem` value of the TCP receive buffer. The reason this sequence is so short and there are no ip-processing calls, is that the IP processing has already happened asynchronously when the packet arrived and is no longer necessary to be done on behalf of the netcat process, therefore it was not traced.[6]

6 Conclusion

The two goals of this work were reached. This text contains an freely available and up to date overview over the Linux TCP networking internals and the ftrace kernel event tracing toolkit. Regarding the third issue, the scope, we tried give a quite general scope, covering no part of the TCP stack explicitly (or solely sending/receiving), so every ascending kernel developer gets some general context of the part functionality he or she wants to change or improve. Nevertheless TCP Networking is already a quite focused topic, so the third issue was not completely solved.

Concerning the decision for using ftrace it must be said that using ftrace solely was no very good idea, since ftrace (or trace-cmd) can not report any information about the function parameters used.[8][5] But knowing what data a function is using (or what pointers to data) is quite important for tracing the way of data. So a combined method of using ftrace and looking up function prototypes in the Linux source code was employed and worked fairly well.

So for generally understanding how these functions work together ftrace and prototype look-up is a reasonable method. But if the goal is fixing errors and knowledge about the concrete values passed to a function get necessary other tools like *kgdb* and *systemtap* are more well-suited.

8. There exists no option or capability in the ftrace documentation regarding function parameters.

Bibliography

[1] Christian Benvenuti. *Understanding Linux network internals*. O'Reilly, Sebastapol, Calif, 2006.

[2] Mike Frysinger. Function tracer guts. Doc-file in Linux source tree: linux/Documentation/trace/ftrace-design.txt.

[3] Johan Kraft, Anders Wall, and Holger Kienle. Trace recording for embedded systems: lessons learned from five industrial projects. In *Proceedings of the First International Conference on Runtime Verification (RV 2010)*. Springer-Verlag (Lecture Notes in Computer Science), November 2010. Original publication is available at www.springerlink.com.

[4] Robert Love. *Linux kernel development*. Addison-Wesley, Upper Saddle River, NJ, 2010.

[5] Steven Rostedt. Ftrace - function tracer. Doc-file in Linux source tree: linux/Documentation/trace/ftrace.txt.

[6] R. Rosen. *Linux Kernel Networking: Implementation and Theory*. Books for professionals by professionals. Apress, 2013.

[7] Steven Rostedt . *Trace-cmd-record(1) Linux Manpage*. 2010.

[8] Andrew S Tanenbaum. Computer networks, 4-th edition. *Ed: Prentice Hall*, 2003.

[9] Wenji Wu, Matt Crawford, and Mark Bowden. The performance analysis of linux networking–packet receiving. *Computer Communications*, 30(5):1044–1057, 2007.